Theoretical Computer Science 118 (1993) 49–65
Elsevier

A speed-up theorem without tape compression

Viliam Geffert

Department of Computer Science, University of P.J. Šafárik, Jesenná 5, 04154 Košice, Slovak Republic

Abstract

Geffert, V., A speed-up theorem without tape compression, Theoretical Computer Science 118 (1993) 49–65.

We shall show that, for each nondeterministic single-tape Turing machine M of time complexity $T(n) \geqslant n^2$ and each $K \geqslant 1$, there exists an equivalent K times faster nondeterministic Turing machine M' writing only zeroes and ones on its tape. In other words, we can obtain the linear speed-up while preserving the binary worktape alphabet. Therefore, nondeterministic single-tape Turing machines do not require tape compression for speeding up.

1. Introduction

The classical speed-up theorem [2] for single-tape Turing machines (both deterministic and nondeterministic) asserts that each $T(n)$-time-bounded Turing machine M, with $T(n) \geqslant n^2$, can be replaced by an equivalent Turing machine M' of time complexity $T'(n) \leqslant T(n)/K$, for each $K \geqslant 1$.

However, the classical proof technique of this theorem requires the number of worktape symbols to be increased because we have to pack several symbols of the original tape alphabet into one symbol of a new alphabet, in order to reduce the number of moves of the tape head. The cardinality of the worktape alphabet grows exponentially in the speed-up factor K.

On the other hand, it is obvious that we can easily replace each Turing machine by a machine using the binary tape alphabet only. The price we pay is a linear slowdown of the computation time.

We shall show that, for each nondeterministic single-tape Turing machine of time complexity $T(n) \geqslant n^2$ and each $K \geqslant 1$, there exists an equivalent K times faster

Correspondence to: V. Geffert, Department of Computer Science, University of P.J. Šafárik, Jesenná 5, 04154 Košice, Slovak Republic. Email: geffert@csearn.bitnet.

0304-3975/93/$06.00

nondeterministic single-tape machine M' writing only two symbols (0 and 1) on its tape, that is, we can reduce the time and cardinality of the tape alphabet simultaneously. For subquadratic $T(n)$ (or for $T(n)$ not satisfying $T(n) \geqslant n^2$), we have only to add a single new symbol to the original tape alphabet. However, our proof technique is based on nondeterminism and cannot be extended to the deterministic case.

2. Speeding up

We shall consider nondeterministic Turing machines with a single semi-infinite read–write worktape, beginning its computation with the input word stored on this tape.

Definition 2.1. A nondeterministic Turing machine is said to be *strongly $T(n)$-time-bounded* if no computation path on any input of length n executes more than $T(n)$ steps. It is *weakly $T(n)$-time-bounded* if, for each accepted input of length n, there exists at least one accepting computation path which executes at most $T(n)$ steps.

Similarly, we can consider strongly and weakly space-bounded nondeterministic Turing machines.

Definition 2.2. A function $T(n)$ is *time-well-countable* if there exists a Turing machine M_T which, starting with the binary-coded value of n on its tape, halts within time $o(T(n))$ and computes the binary-coded value of $T(n)$.

All "normal" functions growing faster than $\log(n) \cdot \log\log(n)$ are time-well-countable. Note that there is a difference between the standard definition of a time-countable function and the definition of a time-well-countable function as given above. (The latter seems to be a little more restrictive, since we do not claim $T(n)$ to be computed in time $O(T(n))$, but rather in time $o(T(n))$.)

We may assume, without loss of generality, that the original machine halts only if the tape head is at the left endmarker. (This modification will slow down the computation by at most a factor of two.)

Further, we do not have to take the number of stationary moves into account (i.e., steps in which the machine does not change the position of its head). These moves can be easily eliminated, which will reduce the computation time.

Theorem 2.3. *For each nondeterministic Turing machine M, there exists an equivalent machine M' not using stationary moves (i.e., the head changes its position in each computation step). Moreover, if M accepts an input w by a computation path d of length d_w, then M' can accept w by a computation path of length $d'_w = d_w - d_w^{ST}$, where d_w^{ST} denotes the number of stationary moves in d.*

We are now ready to show the proof of the main result. We shall describe M' simulating the original machine M by as many stationary moves as possible. The transfer of information between the finite-state control and the worktape will require less time than the original computation of M, since the worktape will be divided into segments so that the lengths of crossing sequences between two adjacent segments will not be too long. More exactly, all segments will be of equal length l (for a fixed constant l), except for the leftmost segment, which will be of length $r \leqslant l$. The value of r will be guessed nondeterministically at the very beginning of the simulation so that the total length of crossing sequences between segments will not exceed $T(n)/l$.

Before passing further, we should show that, for each computation path d executing at most $T(n)$ steps and each l, there does exist $r \leqslant l$ such that the total length of crossing sequences between adjacent tape segments of length l (with the leftmost segment of length r) is at most $T(n)/l$. (A similar argument was also used, for example, in [5, 7, 3] to show the relationship between time and space for single-tape Turing machines.) For each r, l, i, define

d_{rli} = the number of steps in the computation path d such that the head of M crosses the boundary between tape positions $r+il$ and $r+il+1$, no matter in which direction. (The consecutive cells of the tape are numbered from left to right, beginning with zero for the left endmarker.)

Define

$$d_{rl} = \sum_{i=0}^{\infty} d_{rli}.$$

The number d_{rl} denotes the total length of crossing sequences between tape segments. If M is $S(n)$-space-bounded, then

$$d_{rl} = \sum_{i=0}^{S(n)/l} d_{rli}, \tag{1}$$

since $d_{rli}=0$ for each $i \geqslant S(n)/l+1$. It is not too hard to verify that the sum $d_{1l}+\cdots+d_{ll}$ gives (exactly) the total length of the computation path d. This number is at most $T(n)$ for input of length n, which implies that, for at least one $r \leqslant l$, we have

$$d_{rl} \leqslant T(n)/l. \tag{2}$$

First, the machine M' guesses nondeterministically a proper partition of tape into segments (i.e., a proper $r \leqslant l$), computes the length of the input $w = x_1 \ldots x_n$ (modulo l, within its finite-state control), and then prepares its tape for simulation, that is, M' creates binary-coded tape segments for $x_1 \ldots x_r$, $x_{r+1} \ldots x_{r+l}$, $x_{r+l+1} \ldots x_{r+2l}, \ldots$ (The last segment will be filled in by binary-coded blank symbols so that all segments, except for the leftmost one, will be of equal length.) These binary-coded tape segments will be separated on the tape by binary-coded crossing sequence queues (of fixed length), which will be explained later.

Thus, we need $l \cdot c_S$ bits for each tape segment (where c_S is the number of bits needed to code the tape symbols of M, i.e., c_S is a constant dependent only on the cardinality of the original tape alphabet).

Using the power of nondeterminism, M' can simulate M on separate tape segments separately. But M' must also keep track of crossing sequences between adjacent segments, since different tape segments can contain data corresponding to different stages of simulation. Therefore, there are binary-coded crossing sequence queues between each two segments on the tape, and after the rightmost tape segment. The queues between adjacent tape segments are used to store information about the states of the corresponding crossing sequences. The simulation on each particular tape segment is interrupted and resumed so that M' never needs to store more than k states in any crossing sequence queue (where k is a fixed constant). The states are stored and removed from the corresponding crossing sequence queues in the ordinary FIFO manner.

We shall use $2 + k \cdot c_Q$ bits for each crossing sequence queue, where c_Q is a constant dependent only on the number of states of the original machine M. (One binary code is reserved for $v \notin Q$, a "padding state", which is used to fill in the holes if some queues contain less than k states.) The two extra bits are used to code one of the four possible modes of each queue, namely, L, R, L_f and R_f. The L and R modes are used to distinguish which of the two adjacent tape segments is in the more advanced stage of simulation. In addition, L_f and R_f indicate that the simulation on the left (right) tape segment has been completed, i.e., it has reached its final stage. Initially, all queues are empty, in the mode L.

Figure 1 shows an example of a typical situation which may occur during the simulation. The dashed part of the line, representing the movement of the tape head,

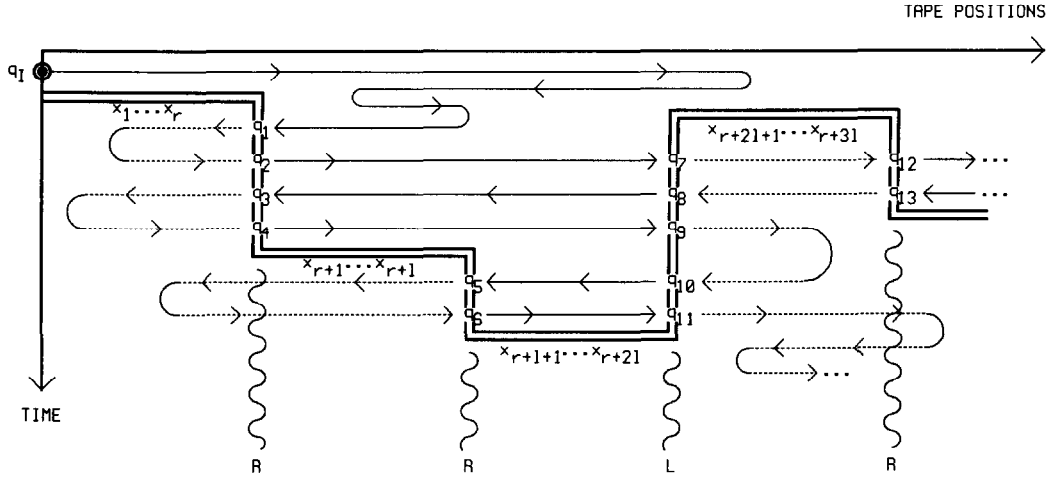

Fig. 1.

shows which moves have not been verified yet and must be simulated in the future. Different tape segments correspond to different time moments of the computation; states of crossing sequences have been guessed nondeterministically. The above situation will be binary-coded on the tape of M' as follows:

$x_1 \dots x_r$	$Rq_1q_2q_3q_4v \dots v$	$x_{r+1} \dots x_{r+l}$	$Rq_5q_6v \dots v$
1st segment $r \cdot c_S$ bits	1st queue $2+k \cdot c_Q$ bits	2nd segment $l \cdot c_S$ bits	2nd queue $2+k \cdot c_Q$ bits

$x_{r+l+1} \dots x_{r+2l}$	$Lq_7q_8q_9q_{10}q_{11}v \dots v$	$x_{r+2l+1} \dots x_{r+3l}$	$R \dots$
3rd segment $l \cdot c_S$ bits	3rd queue $2+k \cdot c_Q$ bits	4th segment $l \cdot c_S$ bits	$\cdots$

Recall that the simulation on particular segments will be interrupted and resumed so that M' never needs to store more than k states in any queue. Therefore, all information about the current segment and its two adjacent queues can be kept within the finite-state control, since segments and queues are of constant lengths.

We are now ready to present the simulation algorithm. Many details and exceptional cases are omitted in order not to obscure the essentials of the algorithm. Some of them require an additional explanation, which will be done later.

Step 1: Prepare the tape for simulation, that is, guess $r \leqslant l$ and create the binary-coded tape segments (of l symbols each, the leftmost one of r symbols only), separated by the binary-coded crossing sequence queues of at most k states (initially empty, in mode L).

Position the tape head at the left endmarker.

Step 2: Load from the tape the data about the next tape segment to the right and also about its two adjacent crossing sequence queues into the finite-state control. This requires the tape head to move $(2+k \cdot c_Q)+(l \cdot c_S)+(2+k \cdot c_Q)$ positions, i.e., one segment and two queues, to the right. (If the head is at the left endmarker, then there is no left queue.)

Step 3: (a) Determine the state q and the position of the tape head at the moment when the last simulation on the current tape segment was interrupted. (For details, see the remark below the algorithm.)

(b) Simulate M (in the finite control of M') on the current tape segment from the moment when the last simulation was interrupted until M crosses a segment boundary. There are now the following cases.

Step 3.1. M has crossed the left boundary, in a state q, and the left crossing sequence queue is in L mode:

(a) Remove one state from the left queue and check whether it is equal to q.

(b) If no, then halt and reject the input, since there has been made a wrong nondeterministic guess during the simulation on the segment lying to the left of the current segment.

(c) If yes, remove one more state from the left queue and carry on the simulation (i.e., go to step 3b; this state is the state the machine M will be in when it returns to the current segment from the left).

(d) If there is no state to be removed from the left queue (the queue is empty), switch the mode of the left queue to R and go to step 3.2.

Step 3.2. M has crossed the left boundary, in a state q, and the left queue is in R mode:

(a) Add q to the left queue.

(b) If the left queue is full (i.e., the queue contains k states) then interrupt the simulation on the current segment and resume the simulation on the next segment to the left. Go to step 4.1.

(c) *The queue is not full.* Guess nondeterministically the state q' which the machine M will be in when it returns to the current tape segment.

(d) Add q' to the left queue and then, using q', carry on the simulation. Go to step 3b.

Steps 3.3 and 3.4. M has crossed the right boundary of the current segment: These cases are analogous to steps 3.1 and 3.2, respectively, but there are the following differences:

- The states are removed (and checked) from the right queue when it is in R mode and they are stored (and guessed) when the queue is in L mode.
- The empty right queue in R mode (all states have been removed) is switched to L mode in step 3.3d (cf. step 3.1d).
- The full right queue in L mode (no more states can be stored in) interrupts the simulation on the current segment (step 3.4b) and the simulation on the next segment to the right is resumed. Go to step 4.2 (cf. step 3.2b).

Step 4.1: Interrupt, write back, and move to the left.

During the simulation (step 3), M' performs only stationary moves, with the tape head positioned after the right queue of the current segment.

By moving the head $(2+k\cdot c_Q)+(l\cdot c_S)+(2+k\cdot c_Q)$ positions to the left, write the current contents of the segment and its two queues back on the tape.

Then move the head $(2+k\cdot c_Q)+(l\cdot c_S)$ more positions (i.e., one segment and one queue) to the left in order to resume the simulation on the next segment to the left. Go to step 2.

Step 4.2: Interrupt, write back, and move to the right.

Write back the segment and its two queues on the tape, by moving the head $(2+k\cdot c_Q)+(l\cdot c_S)+(2+k\cdot c_Q)$ positions to the left. Then move the head $(2+k\cdot c_Q)+(l\cdot c_S)$ positions to the right and go to step 2.

Remark. The state q and the segment boundary from which the simulation is resumed in step 3a depend on the previous steps of the algorithm:
- If step 2 is executed after step 1 (the first simulation on the leftmost segment) then the simulation begins in the initial state q_1 at the left boundary.
- If step 2 is executed after step 4.2 (the previous simulation on the segment to the left) then the state q is removed from the left crossing sequence queue and step 3b starts at the left boundary of the current segment.
- Similarly, if it is executed after step 4.1 (the previous simulation on the segment to the right) then q is removed from the right queue and step 3b starts at the right boundary.

Note that the above algorithm needs k to be odd. For example, if step 2 is executed after step 4.2 then the queue in between is in L mode and full (containing an odd number of states), with the first state corresponding to a computation step moving the tape head to the right. The first state is removed from this queue in step 3a in order to resume the simulation. The further states are removed always in pairs (steps 3.1a and c). Thus, we have to check whether the queue is empty before the execution of step 3.1a, but not before step 3.1c. Similarly, when this queue is switched to R mode (step 3.1d), the states are also stored in pairs (steps 3.2a and d). Note that we check whether the queue is full in step 3.2b, but we are sure it is not full after step 3.2d. Now, the first state that is stored corresponds to a computation step moving the head to the left. This state will be removed by step 3a when the simulation on the left neighboring segment is resumed.

(We could use an algorithm working properly for any k, but this needs a more complicated strategy.)

There are many exceptions requiring an additional explanation:

(A) *The leftmost segment.* If the current segment is the leftmost then the left queue is not used in step 3. Moreover, this segment contains only $r \leqslant l$ symbols and, therefore, M' has to keep the value of r in the finite-state control during the whole computation. (Consequently, steps 2 and 4 must also be modified.)

(B) *Tape extension at the right end.* If the simulation on the rightmost segment is interrupted by going to step 4.2 (the right queue of the rightmost segment is full) then step 2, instead of loading data from the tape into the finite-state control, creates one more tape segment containing l binary-coded blank symbols of M, followed by one more queue, initially empty and in L mode. This situation is detected very easily, since the tape head of M' is positioned at the first blank symbol during the simulation in step 3. The total number of the head moves is equal to the standard case.

(C) *Rejection.* Halt and reject the input if M halts and rejects. (Simulation in step 3b.)

The acceptance is not so simple because the simulation has to be completed on all segments.

(D) *The last state of the crossing sequence.* If in step 3.2a the last state of the crossing sequence has been added to the left queue in R mode then M will never return back to

the current segment because it will halt and accept with the head at the left endmarker. Therefore, M' cannot guess nondeterministically the next state of the crossing sequence in step 3.2c. Instead, since the simulation on the current segment has been completed (this is guessed nondeterministically in step 3.2c), switch the mode of the left queue to R_f, interrupt the simulation immediately and move one segment to the left, no matter whether the left queue is full or not. Go to step 4.1.

Queues in R_f mode are handled in the same way as ordinary queues in R mode, but they are remove-only: any attempt to add a state to such a queue will cause an immediate rejection of the input (wrong guess in the past), because it already contains the last state of the corresponding crossing sequence. Since no more states will be stored in this queue, it can never become full. Therefore, the simulating machine M' will never go to the neighboring segment, with the queue in between in R_f mode. When the R_f queue becomes empty, it can be switched to L_f mode, but only once and for all (exception F, explained later).

(E) *Accepting termination at the left endmarker.* If M' finds that M halts and accepts (which can happen only in step 3b on the leftmost segment, since M always accepts with the head at the left endmarker) then the simulation on the leftmost segment is completed. Now, the right queue must be either in L mode or empty. (A nonempty queue in R/R_f mode indicates that there are some junk states after the last state of the crossing sequence.)

If the right queue is not empty and in R/R_f mode then halt and reject the input. (There was a wrong guess in the past.)

Otherwise, switch the mode of the right queue to L_f, interrupt the simulation and go to step 4.2 (move one segment to the right).

L_f queues are similar to R_f queues (they behave as remove-only L queues), but there are some differences: while an R_f queue can appear anywhere on the tape when the simulation on the next segment to the right has been completed, an L_f queue indicates much more – a successfully terminated simulation on all segments lying to the left. By (E), the first queue is set to L_f mode only when the simulation on the leftmost segment has been finished, and any other queue can be set to this mode only if all queues to the left have already been set to L_f mode.

(F) *The left queue in L_f mode is empty.* If the last state of the crossing sequence has been removed from the left queue in L_f mode (step 3.1a) then M will never return to the current segment and, hence, the simulation on the current segment has been completed. Thus, we cannot remove the next state of the crossing sequence in step 3.1c from the left queue and carry on the simulation. Instead, check whether the *right* queue is either in L mode or empty. (That is, the right queue can be in R/R_f mode only if it is empty.) Reject if this is not the case (junk after the last state of the crossing sequence in the right queue). Otherwise, switch the right queue to L_f mode and go to step 4.2 (move one segment to the right).

A similar action must be taken in step 3a if the left queue loaded by step 2 is empty and in L_f mode. (We shall not resume a computation which has already been

terminated.) Check whether the right queue is in L mode or empty, set it to L_f mode, and then move more to the right. Go to step 4.2.

Thus, L_f is the terminal mode of each queue. Once a left queue becomes empty and is set to L_f mode, the simulating machine M' goes to the right (step 4.2) and never returns because the right queue of the current segment is also set to L_f mode. (But it is not necessarily empty.)

(G) *Acceptance.* When the right queue of the rightmost tape segment is empty and set to L_f mode then *halt and accept* the input, since simulation on all tape segments has been successfully completed.

Verifying the correctness of the above algorithm is straightforward. It can be shown, by induction on the number of tape segments, that if all nondeterministic decisions (steps 3.2c and 3.4c, exception D) correspond to the same accepting computation path of M then M' stops its simulation at the right end of its tape, with all queues in L_f mode and empty. Conversely, it is easy to check that if M' accepts a word w then the guessed crossing sequences correspond to some accepting computation path of M on w.

Next we shall investigate the time required by this simulation. First we consider the case of weakly time-bounded machines, for which case it is sufficient to concentrate on the shortest accepting computation path.

(i) First, step 1 (tape initiation) does not require more than

$$O(n)+2\frac{2+k\cdot c_Q+l\cdot c_S}{l^2}n^2$$

steps. The tape encoding is done from right to left, segment by segment. The length of the tape used does not exceed $(2+k\cdot c_Q+l\cdot c_S)(n/l+2)$ during the initiation phase. The number of the tape head traversals along the tape can be bounded by $2(n/l+2)$, because one traversal is sufficient to encode l symbols of the original input into one more tape segment, to create one more empty queue, and to shift the already encoded part of the tape more to the right (which requires that at most $2+k\cdot c_Q+l\cdot c_S$ characters be kept in the finite control). This gives the time bound as shown above. It can also be easily seen that the first two symbols of the original tape alphabet can be used as zero and one; therefore, we do not have to use any new tape symbols.

(ii) One simulation batch (that is, load data from the tape, simulate, write back on the tape, and move one segment to the left/right (steps 2 and 3, including steps 3.1–3.4 and step 4.1/4.2)) requires at most $5\cdot(2+k\cdot c_Q)+3\cdot(l\cdot c_S)$ nonstationary moves, because the tape head travels five crossing sequence queues and three tape segments along the tape, plus some stationary moves for the simulation in step 3.

(iii) It can be shown that if the length of the crossing sequence between the ith and $(i+1)$st segments is d_{rli} then at most $\lfloor d_{rli}/k \rfloor+2$ simulation batches cross the boundary between these two segments (step 4.1 or step 4.2):

Clearly, each time the simulation on the ith segment (in step 3) is interrupted by going to step 4.2 (resume the simulation on the $(i+1)$st segment), the right queue must

be in L mode and full. (It must contain k states.) It is not necessary to return immediately to the left when the simulation on the $(i+1)$st segment is interrupted, since the machine can move more to the right (step 4.2) and resume/interrupt simulation on further segments to the right.

But each time the machine simulates on the $(i+1)$st segment, the queue between the ith and $(i+1)$st segments is used as the left queue in L mode (states are removed). When all k states are removed, the queue is switched to R mode and then the next k states of the crossing sequence will be guessed nondeterministically and stored in this queue. M' will interrupt the simulation on the $(i+1)$st segment by going to the left (step 4.1) only when this queue is in R mode and full again.

Similarly, M' moves next time from the ith segment to the right only when the k states stored in the queue are removed, the queue is switched to L mode, and the further k states are stored.

Thus, the total number of simulation batches crossing the boundary between the ith and $(i+1)$st segments is at most $\lfloor d_{rli}/k \rfloor + 2$. There can be one or two extra batches when the last state of the crossing sequence has been stored in the queue. The queue is switched to L_f or R_f mode and the simulation on the current segment (the ith or $(i+1)$st, respectively) is stopped immediately. In addition, the empty R_f queue will be switched to L_f mode after successful simulation on all segments lying to the left, by one more extra batch.

(iv) Therefore, by (1) and (2), the total number of simulation batches is at most

$$\sum_{i=0}^{S(n)/l+2} (d_{rli}/k+2) \leqslant 6+\frac{2}{l}\cdot S(n)+\frac{1}{k\cdot l}\cdot T(n).$$

But then, using (i) and (ii), the time complexity of M' is

$$T'(n) \leqslant O(n)+2\frac{2+k\cdot c_Q+l\cdot c_S}{l^2}\cdot n^2+2\frac{5\cdot(2+k\cdot c_Q)+3\cdot(l\cdot c_S)}{l}\cdot S(n)$$

$$+\frac{5\cdot(2+k\cdot c_Q)+3\cdot(l\cdot c_S)}{k\cdot l}\cdot T(n)+\text{STATIONARY\_MOVES}.$$

It can be easily seen that for space we obtain

$$S'(n) \leqslant \frac{(2+k\cdot c_Q)+(l\cdot c_S)}{l}\cdot(S(n)+2).$$

The stationary moves can be eliminated by Theorem 2.3. Now, suppose that $T(n) \geqslant n^2$. Then, for each $e>0$, using the substitution $k=2h+1$ (k should be odd, see the remark), $l=h^2$ for sufficiently large h, and the fact that $n \leqslant \sqrt{T(n)} \in o(T(n))$, we obtain

$$\begin{aligned} T'(n) &\leqslant \frac{e}{2}\cdot T(n)+(6\cdot c_S+e)\cdot S(n), \\ S'(n) &\leqslant (c_S+e)\cdot S(n), \end{aligned} \tag{3}$$

for all but finitely many n's. For machines using strictly less space than time (i.e., with $S(n) \in o(T(n))$), this gives the following theorem.

Theorem 2.4. *For each single-tape nondeterministic Turing machine M of (weak) time complexity $T(n) \geqslant n^2$ and space complexity $S(n) \in o(T(n))$, and each $e > 0$, there exists an equivalent machine M' writing only zeroes and ones on its tape, weakly time- and space-bounded by*

$$T'(n) \leqslant e \cdot T(n)$$

and

$$S'(n) \leqslant (c_S + e) \cdot S(n),$$

respectively, where c_S is the number of bits needed to code the original worktape alphabet.

The assumption that a $T(n)$-time-bounded machine consumes space $S(n) \in o(T(n))$ was crucial in the precise time analysis of the algorithm given above. We shall now show that this assumption is superfluous and can be discarded. The next theorem plays an important role in the following considerations. (For detailed proofs, the reader is referred to [4–7].) The theorem shows the relationship between time and space complexities of nondeterministic single-tape Turing machines.

Theorem 2.5. *Let L be recognized by a weakly $T(n)$-time-bounded single-tape nondeterministic Turing machine M, for some $T(n) \geqslant n^2$. Then L can be recognized by a nondeterministic single-tape Turing machine M' which is weakly time-bounded by $O(T(n))$ and weakly space-bounded by $O(\sqrt{T(n)})$.*

The original version of this theorem in [5] requires $\sqrt{T(n)}$ to be fully space-constructible in time $O(T(n))$. But this assumption has been used only to extend the theorem to strongly time-bounded nondeterministic machines. This is not necessary if we consider weakly time-bounded machines only. Combining Theorems 2.4 and 2.5, we get the following theorem.

Theorem 2.6. *For each single-tape nondeterministic Turing machine M of (weak) time complexity $T(n) \geqslant n^2$ and space complexity $S(n)$, and each $e > 0$, there exists an equivalent machine M' writing only zeroes and ones on its tape, weakly time- and space-bounded by*

$$T'(n) \leqslant e \cdot T(n)$$

and

$$S'(n) \leqslant (c_S + e) \cdot S(n),$$

respectively.

Proof. It is obvious that, by Theorem 2.5, we get a machine satisfying the assumptions of Theorem 2.4. But we have to show that there are no hidden penalties, for example, in the alphabet size. By Theorem 2.5, we have a new machine that is time-bounded by $c \cdot T(n)$ and space-bounded by $c \cdot \sqrt{T(n)}$, for some constant $c \geqslant 1$. There is no loss of generality in assuming that this machine uses the same worktape alphabet as the original machine. (This increases the constants c and c_Q, but not c_S. We can use the binary tape alphabet as well.) But then, by (3), using the speed-up factor e/c instead of e, we can simulate the machine of Theorem 2.5 in time and space

$$T'(n) \leqslant e/2 \cdot T(n) + (6 \cdot c_S + e) \cdot c \cdot \sqrt{T(n)}$$

and

$$S'(n) \leqslant (c_S + e) \cdot c \cdot \sqrt{T(n)},$$

respectively. This gives $T'(n) \leqslant e \cdot T(n)$ because $\sqrt{T(n)} \in o(T(n))$.

The situation is more complicated if we want to satisfy both the time and space bounds. For space we can obtain

$$S'(n) \leqslant \frac{1}{6 \cdot c_S + e} \cdot \frac{e}{2} \cdot T(n)$$

since

$$\sqrt{T(n)} \leqslant \frac{1}{(c_S + e) \cdot c} \cdot \frac{1}{6 \cdot c_S + e} \cdot \frac{e}{2} \cdot T(n)$$

for each sufficiently large n. Clearly, if

$$\frac{1}{6 \cdot c_S + e} \cdot \frac{e}{2} \cdot T(n) \leqslant S(n)$$

then $S'(n) \leqslant S(n)$ and we are done. On the other hand, if

$$S(n) < \frac{1}{6 \cdot c_S + e} \cdot \frac{e}{2} \cdot T(n)$$

then *the simulation of the original machine M* gives the claimed time and space bounds, by (3). Therefore, we construct a new machine that guesses nondeterministically (at the very beginning, never verifying its guess) which way is more efficient, i.e., to simulate the original machine, or the machine of Theorem 2.5. Recall that for weakly bounded machines it is sufficient to concentrate on the shortest computation path. This trick need not be used if $S(n)$ is either "sufficiently small" or "sufficiently large", i.e., if $S(n) \in o(T(n))$ (simulate the original machine M) or $\sqrt{T(n)} \in o(S(n))$ (simulate the machine of Theorem 2.5). □

3. Some variants

Note that if we do not want to use the binary worktape alphabet, but rather the worktape alphabet of the original machine M, or if the original machine already uses the binary worktape alphabet, then we have the following theorem.

Theorem 3.1. *For each single-tape nondeterministic Turing machine M (weakly) time- and space-bounded by $T(n) \geqslant n^2$ and $S(n)$, respectively, and each $e_1, e_2 > 0$, there exists an equivalent machine M' using the same worktape alphabet, weakly time- and space-bounded by*

$$T'(n) \leqslant e_1 \cdot T(n)$$

and

$$S'(n) \leqslant (1 + e_2) \cdot S(n),$$

respectively. That is, any time reduction can be obtained at the cost of an arbitrarily small additional space.

But a minor problem arises here since the above algorithm was used to code blank symbols of M on the tape so that the rightmost segment was also of length l. For example, this would give $c_S = 2$ for the binary tape alphabet. We can easily avoid this problem, for example, by keeping the rightmost segment and the rightmost queue in the finite-state control or by allowing the rightmost segment to be shorter. Then the blank symbols of M need not be coded or stored on tape.

Using a single new symbol, we can extend the above theorem to nonlinear time bounds not necessarily satisfying $T(n) \geqslant n^2$. Moreover, no extra space is needed in this case.

Theorem 3.2. *For each single-tape nondeterministic Turing machine M using an H-letter worktape alphabet ($H \geqslant 2$), time-bounded by $T(n)$, satisfying $\lim_{n \to \infty} n/T(n) = 0$, and space-bounded by $S(n)$, and each $e > 0$, there exists an equivalent machine M' using an $(H+1)$-letter worktape alphabet, weakly time- and space-bounded by*

$$T'(n) \leqslant e \cdot T(n)$$

and

$$S'(n) \leqslant S(n),$$

respectively.

Proof. The assumption that $T(n) \geqslant n^2$ has been used only twice.

First, the original machine M simulated by the algorithm described above was supposed to use strictly less space than time, i.e., with $S(n) \in o(T(n))$. For machines not

satisfying this assumption, we had to use Theorem 2.5, in order to obtain a machine which was time-bounded by $T^0(n) \in O(T(n))$ and space-bounded by $S^0(n) \in O(\sqrt{T(n)})$, i.e., with $S^0(n) \in o(T^0(n))$.

The assumption that $T(n) \geqslant n^2$ was used in the proof of Theorem 2.5. It is not very hard to see that, using a slightly modified proof of Theorem 2.5, we can obtain a machine which is time-bounded by $T^0(n) \in O(T(n))$ but space-bounded by $S^0(n) \in O(\max(n, \sqrt{T(n)}))$, even if $T(n)$ does not satisfy $T(n) \geqslant n^2$ for some n's. But if $T(n)$ is nonlinear, i.e, if $\lim_{n\to\infty} n/T(n) = 0$, then we still have $S^0(n) \in o(T^0(n))$.

Second, step 1 of the above algorithm (tape initiation) requires quadratic time, because we have to create some room between each of the two tape segments where the crossing sequence queues can be placed.

Clearly, using the first two letters of the original worktape alphabet as zero and one, we can code each crossing sequence queue containing k states (and its mode) by $2 + k \cdot c_Q$ symbols, and, hence, we can code one queue and one tape segment consisting of l symbols by $2 + k \cdot c_Q + l$ symbols, using the original H-letter worktape alphabet.

The strings of length $2 + k \cdot c_Q + l$ over any H-letter alphabet can be unambiguously coded by strings of length l over any $(H+1)$-letter alphabet if $H^{2+k\cdot c_Q+l} \leqslant (H+1)^l$. This condition is satisfied for properly chosen k and l. Recall that $k = 2h+1$ and $l = h^2$ for some sufficiently large h. Because

$$\lim_{h\to\infty} \frac{(H^{c_Q \cdot 2})^h}{(1+1/H)^{h\cdot h}} = 0,$$

and, therefore,

$$\frac{(H^{c_Q \cdot 2})^h}{(1+1/H)^{h\cdot h}} \leqslant \frac{1}{H^{2+c_Q}} \quad \text{for sufficiently large } h,$$

we have $H^{2+(2h+1)\cdot c_Q + h^2} \leqslant (H+1)^{h^2}$. Thus, we can encode one queue for k states and one segment of l symbols into a string of length l, using an $(H+1)$-letter alphabet. But then step 1 can be executed in time $O(n)$; it will be a single traversal along the tape from left to right. (Steps 2 and 4.1/4.2 must also be modified, since data must be decoded or encoded when information is loaded from or written back on the tape.) Clearly, we need no extra space, i.e., $S'(n) \leqslant S(n)$. □

Recall that all the above arguments hold only for the shortest accepting computation paths and, therefore, only for weakly time-bounded machines, since there can also be longer accepting computation paths (for example, if M' does not guess a proper $r \leqslant l$). Moreover, M' can even enter an infinite loop (due to a wrong nondeterministic decision adding a wrong state to some queue). These longer computations may be halted [5] if $T(n)$ is time-well-countable (cf. Definition 2.2). Then the machine can count the simulated steps.

Theorem 3.3 (Liśkiewicz and Loryś [5]). *Let L be recognized by a weakly $T(n)$-time-bounded single-tape nondeterministic Turing machine M, for some time-well-countable function $T(n) \geqslant n^2$. Then L can be recognized by a strongly time-bounded machine M' in time $T'(n) \leqslant c \cdot T(n)$, for some constant c.*

The proof in [5] is based on the observation that a nondeterministic machine M' can, in time $O(T(n))$, simulate M, count the number of simulated steps, and halt each computation path trying to execute more than $T(n)$ steps. It uses a special counter, the so-called Fürer's counter, which was originally constructed to refine the time hierarchy for multitape deterministic Turing machines [1].

The counter has a form of a full binary tree in which each node contains a quaternary digit. If, for each node v, $h(v)$ denotes the height of v (i.e., the distance to the leaves) and $d(v)$ denotes the digit stored in v, then the content of the counter is the number $\sum d(v) \cdot 4^{h(v)}$. If we want to subtract one from the counter, we may subtract one from any of the leaves or, if the chosen leaf v contains zero, then we shall find the nearest ancestor u with a nonzero digit, subtract one from it and put the digit 3 to all nodes along the path from v to u. To operate the counter, we shall use some additional tracks on the worktape (see Fig. 2):

Track 1: Content of the counter (digits 0, 1, 2 or 3).

Track 2: Direction to the father (L – to the left, R – to the right, T – the top of the tree).

Track 3: Auxiliary (among others, to compute the distances between nodes and their fathers when traversing along the tree).

Track 4: The worktape of the original machine M.

The simulating machine, each time one step of the original machine is executed, subtracts one from the nearest leaf of the tree. If the counter is "exhausted" (i.e., if there is no node with a nonzero digit along the path from the nearest leaf to the root), then the simulation is interrupted and the input is rejected.

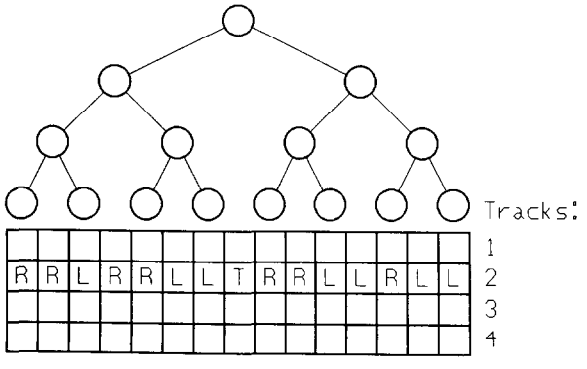

Fig. 2.

A careful analysis of the proof in [5] reveals that the constant $c \leqslant 300$ if we simulate this way a machine M using the binary tape, by a machine which also uses the binary tape (i.e., the tape with Fürer's counter is binary-coded). That is, the constant factor c is not dependent on the number of states or program instructions of M. More exactly, having the binary-coded Fürer's counter of length $O(\sqrt{T(n)})$ on the tape, we can simulate the first t steps and count down in time $t' \leqslant 300t$. Moreover, if the counter contains the value of $T(n)$, then M' will never try to simulate more than $T(n)$ steps of M along any computation path. (The counter must be exhausted after $T(n)$ steps.)

But there are several ways of assigning digits to the nodes of the tree so that the content of the counter is equal to $T(n)$. The value of $T(n)$ has to be "distributed properly" along the tape in the counter so that the simulating machine M' does not interrupt too early, not having simulated $T(n)$ steps. Therefore, the counter is initiated nondeterministically. Assign digits 0, 1, 2 or 3 to each node and then verify whether the content of the counter is equal to $T(n)$. This can be done in time $O(\sqrt{T(n)} \cdot \log(T(n)))$. (In the original proof [5], the initial value of each node was 3; this is not sufficient here.) Then the initial value of the counter and, hence, also the constant c would be dependent on the space complexity of M and also on c_S.

Further, the original assumption [5] that $\sqrt{T(n)}$ is fully space-constructible in time $d \cdot T(n)$, for some constant d, must be replaced by the time-well-countability of $T(n)$.

M' begins its computation by computing the value of $T(n)$ and by marking $O(\sqrt{T(n)})$ space on the tape (which takes $d \cdot T(n)$ time in the original proof). Then the constant c would be greater than d. Moreover, the machine M_T (cf. Definition 2.2) computing $T(n)$ may use a larger tape alphabet. Hence, we have to code M_T's worktape symbols by several symbols of the binary tape alphabet. This would slow down the initial computation by another constant factor. These problems can be resolved if $T(n)$ can be computed in time $o(T(n))$. Then the constant factor associated with the initial computation of $T(n)$ does not play an important role for large n.

By Theorem 3.3, we can replace each weakly $T(n)$-time-bounded machine by an equivalent strongly time-bounded machine. This will slow down the computation by at most a factor of 300. But we may lose space efficiency if the original machine uses less than $\sqrt{T(n)}$ space.

Theorem 3.4. *Let $T(n) \geqslant n^2$ be a time-well-countable function. Then for each (strongly or weakly) $T(n)$-time-bounded single-tape nondeterministic Turing machine and each $e > 0$ there exists an equivalent strongly time-bounded machine writing only zeroes and ones on its tape of time complexity $T'(n) \leqslant e \cdot T(n)$.*

Thus, we see that the tape compression is not necessary for speeding up on nondeterministic single-tape Turing machines, even if they are strongly time-bounded. Taking Theorem 3.4 into consideration, the following open problems arise:

- It is not known whether tape compression is necessary for deterministic machines.
- Is it possible to extend the above result to two-tape nondeterministic machines?

Acknowledgment

This research was supported by grant SPZV I-1-5/8.

References

[1] M. Fürer, The tight deterministic time hierarchy, in: *Proc. 14th Ann. ACM Symp. on Theory of Computing* (1982) 8–16.

[2] J. Hartmanis and R.E. Stearns, On the computational complexity of algorithms, *Trans. Amer. Math. Soc.* **117** (1965) 285–306.

[3] O.H. Ibarra and S. Moran, Some time–space tradeoff results concerning single-tape and off-line TM's, *SIAM J. Comput.* **12** (1983) 388–394.

[4] M. Li, H. Neuféglise, L. Torenvliet and P. van Emde Boas, On space efficient simulations, ITLI prepublication series for computation and complexity theory CT-89-03, Univ. of Amsterdam, 1989.

[5] M. Liśkiewicz and K. Loryś, Fast simulations of time-bounded one-tape Turing machines by space-bounded ones, *SIAM J. Comput.* **19** (1989) 511–521.

[6] M. Liśkiewicz and K. Loryś, Two applications of Fürer's counter to one-tape nondeterministic TM's, in: *Proc. MFCS '88*, Lecture Notes in Computer Science, Vol. 324 (Springer, Berlin, 1988) 445–453.

[7] L. Torenvliet and P. van Emde Boas, A note on time and space, in: *Proc. of Computing Science in the Netherlands, CSN '87*, SION, Amsterdam (1987) 225–234.